Documentation for Geometry.h and Geometry.c

Steven Andrews, © 2006
See the document “LibDoc” for general information about this and other libraries.

Header file

```
#ifndef __Geometry_h
#define __Geometry_h

// Normal
double Geo_LineNormal(double *pt1,double *pt2,double *ans);
double Geo_LineNormal3D(double *pt1,double *pt2,double *point,double *ans);
double Geo_TriNormal(double *pt1,double *pt2,double *pt3,double *ans);

// Point in
int Geo_PtInTriangle(double *pt1,double *pt2,double *pt3,double *norm,double
     *test);
int Geo_PtInSlab(double *pt1,double *pt2,double *test,int dim);
int Geo_PtInSphere(double *test,double *cent,double rad,int dim);

// To Rect
void Geo_Semic2Rect(double *cent,double rad,double *outvect,double *r1,double
     *r2,double *r3);
void Geo_Hemis2Rect(double *cent,double rad,double *outvect,double *r1,double
     *r2,double *r3,double *r4);
void Geo_Cyl2Rect(double *pt1,double *pt2,double rad,double *r1,double
     *r2,double *r3,double *r4);

// Cross
double Geo_LineXLine(double *l1p1,double *l1p2,double *l2p1,double *l2p2,double
     *crss2ptr);
double Geo_LineXSphs(double *pt1,double *pt2,double *cent,double rad,int
     dim,double *crss2ptr);
double Geo_LineXCyl2s(double *pt1,double *pt2,double *cp1,double *cp2,double
     *norm,double rad,double *crss2ptr);
double Geo_LineXCyls(double *pt1,double *pt2,double *cp1,double *cp2,double
     rad,double *crss2ptr);

// Cross aabbb
int Geo_LineXaabb2(double *pt1,double *pt2,double *norm,double *bpt1,double
     *bpt2);
int Geo_TriXaabb3(double *pt1,double *pt2,double *pt3,double *norm,double
     *bpt1,double *bpt2);
int Geo_RectXaabb2(double *r1,double *r2,double *r3,double *bpt1,double *bpt2);
int Geo_RectXaabb3(double *r1,double *r2,double *r3,double *r4,double
     *bpt1,double *bpt2);
int Geo_CircleXaabb2(double *cent,double rad,double *bpt1,double *bpt2);
int Geo_SphsXaabb3(double *cent,double rad,double *bpt1,double *bpt2);
int Geo_CylisXaabb3(double *pt1,double *pt2,double rad,double *bpt1,double
     *bpt2);
```

```
// Approx. cross aabb
int Geo_SemicXaabb2(double *cent,double rad,double *outvect,double *bpt1,double
     *bpt2);
int Geo_HemisXaabb3(double *cent,double rad,double *outvect,double *bpt1,double
     *bpt2);
int Geo_CylsXaabb3(double *pt1,double *pt2,double rad,double *bpt1,double
     *bpt2);

#endif
```

Includes: `"Geometry.h"`, `“math2.h”`.
Example program: `Smoldyn`

History: Started 2/06. Major rewrite and additions 12/06 and 2/07; included in this was a complete switch from floats to doubles.

Bugs

Several of the functions that check crossings of 3-D objects with others (aabbs in particular) ignore potential separation planes, so they report crossings when there aren’t any. These planes are defined by edges on two different objects. These need to be fixed.

Description

These functions do a variety of things that are useful for 1-D, 2-D, and 3-D geometry manipulations. A few functions do *n*-D geometry, but those are rare. Its sole current use is in Smoldyn.

Functions do not change input data arrays. Output arrays are written to but never read from. In general, it is permissable to use the same input array for multiple inputs, but every output array needs to be distinct from each other, and from each input array.

In general, functions include all boundaries as part of the region when testing whether a point is in a region or not, or whether two regions overlap. An axis-aligned bounding box, called an aabb, has its low corner $(x_{\min},y_{\min},z_{\min})$ at `bpt1` and its high corner $(x_{\max},y_{\max},z_{\max})$ at `bpt2`.

The functions that test whether two objects intersect often make use of the separating axis theorem. However, they also often use my own methods.

Math

Several functions use the cross-product of two vectors. As a reminder, $\mathbf{c} = \mathbf{a}\times\mathbf{b}$ is:

$$c_x = a_y b_z - a_z b_y$$
$$c_y = a_z b_x - a_x b_z$$
$$c_z = a_x b_y - a_y b_x$$

Here is my understanding of the separating axis theorem. Consider two convex polygons, in 2-D. A line segment is included as well, where this can be seen as a two-sided polygon with 0 area. If the objects do not cross, then there must a be at least one infinite line that separates them. One of these lines will be parallel and adjacent to one of the sides of one of the polygons. To check for crossing, (1) choose a polygon edge on object A, (2) find its outward normal, which does not have to normalized, (3) project a vertex of the test edge onto this normal by taking the dot product of the vertex and the normal vector, (4) project all vertices of polygon B onto this normal in the same way, (5) the objects do not cross if all projected values of object B are larger than the projected value of the test edge. Repeat for all edges of both objects; if all projections fail, then the objects must cross. If two edges are parallel, they will have opposite normal vectors and some steps can be saved.

In three dimensions, if two convex polyhedra do not cross, then there must be an infinite plane that separates them. Before, I thought that non-crossing polyhedra necessarily implied a separating plane that is parallel to one of the faces on one of the objects. Now I see that that is sufficient to prove separation, but that other potential separating planes need to be checked as well. Other planes to test are those that are parallel to pairs of edges, with one edge from A and one from B. It is not necessary to test pairs of edges within A, or within B, that do not define a face.

For the line crossing sphere function, here is the math. Consider a circle with its center at the origin and radius *R*. A line segment goes from **r** to **s**. If it crosses the circle, a crossing point will be denoted **x** (it may cross 0, 1, or 2 times, and also the crossing point(s) may be on the line but not in the line segment). The relative position of the crossing point on the line, where **r** defines 0 and **s** defines 1, is *p*. Because the crossing point is on the circle and the center is the origin, $\mathbf{x}^2 = R^2$. Also, the crossing point can be given by interpolation between the ends: $\mathbf{x} = (1-p)\mathbf{r} + p\mathbf{s}$. These are set equal to each other and solved for *p*.

$$R^2 = \left[(1-p)\mathbf{r} + p\mathbf{s}\right]^2$$
$$0 = p^2(\mathbf{s}-\mathbf{r})^2 + p\left[2\mathbf{r}\cdot(\mathbf{s}-\mathbf{r})\right] + \mathbf{r}^2 - R^2$$

These lead to *p* being the solution of the quadratic equation with coefficients

$$a = (\mathbf{s}-\mathbf{r})^2$$
$$b = 2\mathbf{r}\cdot(\mathbf{s}-\mathbf{r})$$
$$c = \mathbf{r}^2 - R^2$$

If b^2-4ac is positive, there are two solutions; if it is negative, there are no solutions; and if it is zero, there is one solution. The number of solutions gives the number of crossings, and the values give the crossing positions. Exactly the same math works for spheres and higher dimension spheres.

For the line crossing line function, here is the math. Line 1 goes from **c** to **d**. Line 2 goes from **r** to **s**. Where they cross, a point along line 1 equals a point along line 2: **c** +

$a(\mathbf{d}-\mathbf{c}) = \mathbf{r} + b(\mathbf{s}-\mathbf{r})$, where a is the position along line 1 and b is the position along line 2. This is in two dimensions, so we have two equations and two unknowns. The math is a bit lengthy but results in the equations:

$$a = \frac{(r_x - c_x)(s_y - r_y) - (r_y - c_y)(s_x - r_x)}{(d_x - c_x)(s_y - r_y) - (d_y - c_y)(s_x - r_x)}$$

$$b = \frac{(r_x - c_x)(d_y - c_y) - (r_y - c_y)(d_x - c_x)}{(d_x - c_x)(s_y - r_y) - (d_y - c_y)(s_x - r_x)}$$

The line segments cross if $0 \le a \le 1$ and $0 \le b \le 1$, where the "or equal" portions of these inequalities depend on what ends are included or excluded. Since a and b have the same denominators, both will have zero denominators for the same situations, which occurs when then lines are parallel or when $\mathbf{c} = \mathbf{d}$ or $\mathbf{r} = \mathbf{s}$.

Code documentation

Externally accessible functions

Normal functions

`double Geo_LineNormal(double *pt1,double *pt2,double *ans);`

Finds the 2-D unit normal for line segment that goes from `pt1` to `pt2` (all 2-D) and puts it in `ans`. The result vector is perpendicular to the line segment and points to the right, for travel from `pt1` to `pt2`. If `pt1` and `pt2` are equal, the unit normal points towards the positive *x*-axis. Returns the length of the line segment from `pt1` to `pt2`.

`double Geo_LineNormal3D(double *pt1,double *pt2,double *point,double *ans);`

Finds the 3-D unit normal for line that includes `pt1` and `pt2`, and that includes the point `point`, and puts it in `ans`. The result vector is perpendicular to the line. Returns the perpendicular distance between the line and `point`. To decrease round-off error, this function calculates the result using `pt1` as a basis point, and then recalculates the result using the new point. If `pt1` and `pt2` are the same, this returns the normalized vector from `pt1` to `point`. If `point` is on the line that includes `pt1` and `pt2`, this returns the perpendicular unit vector this is in the *x*,*y*-plane and that points to the right of the projection of the line in the *x*,*y*-plane, if possible; if not, it returns the unit *x*-vector which, again, is perpendicular to the line.

`double Geo_TriNormal(double *pt1,double *pt2,double *pt3,double *ans);`

Finds the 3-D unit normal for the triangle that is defined by the 3-D points `pt1`, `pt2`, and `pt3` and puts it in `ans`. If one looks at the triangle backwards along the unit normal, the three points show counterclockwise winding; *i.e.* the right-hand rule for the points in sequence yields the direction of the unit normal. The triangle area is returned. If the area is zero, `ans` is returned in the *x*,*y*-plane. This finds the normal and the area using the cross-product of the first two triangle edges.

Point in functions

`int Geo_PtInTriangle(double *pt1,double *pt2,double *pt3,double *norm,double *test);`

Tests to see if the 3-D point `test` is inside the triangle defined by `pt1`, `pt2`, and `pt3`, and which has normal vector `norm`. `norm` does not have to be normalized. Typically, `test` will be in the plane of the triangle, but this is not required; if it isn't, this determines whether `test` is in the trianglular column that is defined by the other points and perpendicular to the plane of the triangle. The function returns 0 if not and 1 if so. If `test` is on a triangle boundary, 1 is returned, although round-off errors often make the boundary imperfectly defined. The method used here is to find the cross product of each triangle edge with the vector that goes from the second point on that edge to `test`; the dot product of that result with the triangle normal is positive if `test` is inside the triangle.

`int Geo_PtInSlab(double *pt1,double *pt2,double *test,int dim);`

Tests to see if the point test is in the slab of space between `pt1` and `pt2`, inclusive, which works for all dimensions. The slab of space is defined to have its boundaries perpendicular to the line that includes `pt1` and `pt2`.

`int Geo_PtInSphere(double *test,double *cent,double rad,int dim);`

Tests to see if the `dim`-D point `test` is inside the sphere centered about `cent` with radius `rad`. This works in all dimensions. The boundary is considered to be part of the sphere.

To Rect functions

`void Geo_Semic2Rect(double *cent,double rad,double *outvect,double *r1,double *r2,double *r3);`

Calculates the smallest non-axis-aligned rectangle that encloses the semicircle that has center at `cent`, radius `rad`, and outward pointing normalized vector `outvect`. The rectangle is returned with the two perpendicular edges extending from `r1` to `r2`, and `r1` to `r3`. `r1` is an end of the semicircle, on the right side of `outvect`, `r2` is the other end, and `r3` is behind `r1`, not on the semicircle.

`void Geo_Hemis2Rect(double *cent,double rad,double *outvect,double *r1,double *r2,double *r3,double *r4);`

Calculate the smallest non-axis-aligned rectangle that encloses the hemisphere that has center at `cent`, radius `rad`, and outward pointing normalized vector `outvect`. The rectangle is returned with the three perpendicular edges extending from `r1` to `r2`, `r1` to `r3`, and `r1` to `r4`. `r1`, `r2`, and `r3` are in the plane of the opening of the hemisphere and `r4` is behind `r1`; no points contact the surface.

`void Geo_Cyl2Rect(double *pt1,double *pt2,double rad,double *r1,double *r2,double *r3,double *r4);`

Calculate the smallest non-axis-aligned rectangle that encloses the cylinder whose axis extends from `pt1` to `pt2`, and has radius `rad`. The rectangle is returned with the three perpendicular edges extending from `r1` to `r2`, `r1` to `r3`, and `r1` to `r4`. `r1`, `r2`, and `r3` are in the plane of the end of the cylinder around `pt1` and `r4` is in the plane of the end around `pt2`, behind `r1`; no points contact the surface.

X (cross) functions

`double Geo_LineXLine(double *l1p1,double *l1p2,double *l2p1,double *l2p2,double *crss2ptr);`

Returns the point at which a line segment that goes from `l1p1` to `l2p2` crosses another which goes from `l2p1` to `l2p2`. These are in 2 dimensions. The returned value is the distance along line 1 where they cross. If `crss2ptr` is not `NULL`, then it is returned pointing to the distance along line 2 where the crossing is. Distances are between 0 and 1 for crossing within the line segment and other values for other crossing positions on the infinite lines. If the two points that define a line segment are the same, for either line, or if the two lines are parallel, the function returns `NaN` in both the returned values and `crss2ptr`.

`double Geo_LineXSphs(double *pt1,double *pt2,double *cent,double rad,int dim,double *crss2ptr);`

Returns the points at which a line crosses a sphere surface. This function works in any dimension `dim`, from1 upwards. The line segment is defined by `pt1` and `pt2`, the sphere center is at `cent`, and the radius is `rad`. The infinite line may cross the sphere surface once, twice, or not at all. In the first two cases, one result is returned normally and the other is returned in `crss2ptr` if the pointer is supplied (send in `NULL` if this answer is unwanted). In the last case, both solutions are `NaN`. The valid solutions that are returned are the points on the line defined with `pt1` as 0 and `pt2` as 1. If only one solution is between 0 and 1, it is returned normally and the other is returned in `crss2ptr`; otherwise, the smaller solution is returned normally and the larger is returned in `crss2ptr`.

`double Geo_LineXCyl2s(double *pt1,double *pt2,double *cp1,double *cp2,double *norm,double rad,double *crss2ptr);`

Determines if the infinite line defined by 2-dimensional points `pt1` and `pt2` crosses the infinite 2-dimensional “cylinder” surface that has an axis that includes points `cp1` and `cp2`, has a unit right-side normal `norm`, and has radius `rad`. This cylinder is really two parallel lines. If the line crosses the cylinder surface at all, it will cross twice. If only one solution is between 0 and 1, it is returned normally and the other is returned in `crss2ptr`; otherwise, the smaller solution is returned normally and the larger is returned in `crss2ptr`. If the line does not cross the cylinder surface, both values will be `NaN`.

`double Geo_LineXCyls(double *pt1,double *pt2,double *cp1,double *cp2,double rad,double *crss2ptr);`

Determines if the infinite line defined by points `pt1` and `pt2` crosses the infinite cylinder surface that has an axis that includes points `cp1` and `cp2` and has radius `rad`.

This is in 3 dimensions. The infinite line may cross the cylinder surface once, twice, or not at all. In the first two cases, one result is returned normally and the other is returned in `crss2ptr` if the pointer is supplied (send in `NULL` if this answer is unwanted). In the last case, both solutions are `NaN`. The valid solutions that are returned are the points on the line defined with `pt1` as 0 and `pt2` as 1. If only one solution is between 0 and 1, it is returned normally and the other is returned in `crss2ptr`; otherwise, the smaller solution is returned normally and the larger is returned in `crss2ptr`.

Xaabb functions

`int Geo_LineXaabb2(double *pt1,double *pt2,double *norm,double *bpt1,double *bpt2);`

Tests to see if a line segment crosses an axis-aligned boundary-box (aabb), all in 2-D. The line extents from `pt1` to `pt2` and has normal vector `norm`; `norm` does not have to be normalized. The aabb is defined by the low coordinates `bpt1` and the high coordinates `bpt2`. Returns 0 if they do not cross and 1 if so.

`int Geo_TriXaabb3(double *pt1,double *pt2,double *pt3,double *norm,double *bpt1,double *bpt2);`

Tests to see if a triangle intersects an aabb, all in 3-D. The triangle is defined by `pt1`, `pt2`, `pt3`, and its normal vector `norm`; `norm` does not have to be normalized. The entire triangle area is considered, and not just the perimeter. The aabb is defined by the low coordinates `bpt1` and the high coordinates `bpt2`. Returns 1 for crossing and 0 for not.

`int Geo_RectXaabb2(double *r1,double *r2,double *r3,double *bpt1,double *bpt2);`

Tests to see if the 2-D rectangle, which is not necessarily axis-aligned, intersects a 2-D aabb, where the whole rectangle area is considered and not just the perimeter. The rectangle has reference point `r1`; `r2` is in one direction from the `r1` while `r3` is in the perpenicular direction. In the rectangle coordinates, `r1` is the origin, `r2` defines the local x-axis, and `r3` defines the local y-axis. Returns 1 for crossing a 0 for not. The aabb boundaries are as usual.

`int Geo_RectXaabb3(double *r1,double *r2,double *r3,double *r4,double *bpt1,double *bpt2);`

Tests to see if the 3-D rectangle (perpendicular parallelpiped, technically), which is not necessarily axis-aligned, intersects a 3-D aabb, where the whole rectangle volume is considered and not just the perimeter. The rectangle has reference point `r1`; `r2` is in one direction from the `r1`, `r3` is in a perpendicular direction, and `r4` is in the other perpendicular direction. Returns 1 for crossing a 0 for not. The aabb boundaries are as usual.

`int Geo_CircleXaabb2(double *cent,double rad,double *bpt1,double *bpt2);`

Tests to see if the perimeter of a circle crosses an aabb, all in 2-D. The circle is defined by its center location `cent` and its radius `rad`, while the aabb is defined by the low coordinates `bpt1` and the high coordinates `bpt2`. Returns 1 for crossing and

0 for not. The tests are: 1) if SAT on aabb fails, returns 0; 2) if all corners are inside circle, returns 0; 3) if any corners, but not all, are inside circle, returns 1; and 4) if circle center is within the axis-aligned stripes defined by the aabb, returns 1. Tests 2 and 3 treat all box corners as being included with the box.

`int Geo_SphsXaabb3(double *cent,double rad,double *bpt1,double *bpt2);`

Tests to see if the surface of a sphere cross an aabb, all in 3-D. The sphere is defined by its center location `cent` and its radius `rad`, while the aabb is defined by the low coordinates `bpt1` and the high coordinates `bpt2`. Returns 1 for crossing and 0 for not. See `Geo_CircleXaabb2` for the tests that are carried out.

`int Geo_CylisXaabb3(double *pt1,double *pt2,double rad,double *bpt1,double *bpt2);`

Tests to see if an infinite cylindrical shell crosses an aabb, in 3-D. The axis of the radius `rad` cylinder includes the points `pt1` and `pt2`, but is infinitely long. This returns 1 for crossing and 0 for not. The method that this uses is to project along the cylinder axis so that the cylinder becomes a circle and the aabb becomes 8 2-D points that are connected by 12 edges. Then, several tests are run to see if the circle and the box shadow overlap: 1) if SAT on any edge direction fails, returns 0; 2) if all corners are inside the circle, returns 0; 3) if any corners, but not all, are inside the circle, returns 1; 4) if any edge crosses the circle, returns 1; 5) if a line that crosses the center of the circle does not intersect any edges, in at least one direction away from the center, returns 0. While these many tests create a long function, I think that it is reasonably efficient. It should also be accurate.

Approximate Xaabb functions

`int Geo_SemicXaabb2(double *cent,double rad,double *outvect,double *bpt1,double *bpt2);`

Tests to see if a semicircle perimeter crosses an aabb, in 2-D. This can return false positives, described below. `cent` is the semicircle center, `rad` is the semicircle radius, and `outvect` is the normalized outward pointing vector that defines the direction of the semicircle opening. The aabb is defined by `bpt1` and `bpt2`. This returns 1 for crossing and 0 for not. This returns 1 if both the full circle perimeter crosses the aabb, and the minimal rectangle that encloses the semicircle also crosses the aabb. False positive results can occur if the aabb extends into the semicircle interior but does not cross the semicircle.

`int Geo_HemisXaabb3(double *cent,double rad,double *outvect,double *bpt1,double *bpt2);`

Tests to see if a hemisphere shell crosses an aabb, in 3-D. This can return false positives, described below. `cent` is the hemisphere center, `rad` is the hemisphere radius, and `outvect` is the normalized outward pointing vector that defines the direction of the hemisphere opening. The aabb is defined by `bpt1` and `bpt2`. This returns 1 for crossing and 0 for not. This returns 1 if both the full spherical shell crosses the aabb, and the minimal rectangle that encloses the hemisphere also crosses the aabb. False positive results can occur if the aabb extends into the hemisphere interior but does not cross the hemisphere.

```
int Geo_CylsXaabb3(double *pt1,double *pt2,double rad,double *bpt1,double
     *bpt2);
```

Tests to see if a cylindrical shell crosses an aabb, in 3-D. This can return false positives, described below. The cylinder axis extends from `pt1` to `pt2` and `rad` is the cylinder radius. The aabb is defined by `bpt1` and `bpt2`. This returns 1 for crossing and 0 for not. This returns 1 if both the infinite length cylindrical shell crosses the aabb, and the minimal rectangle that encloses the cylinder also crosses the aabb. False positive results can occur if the aabb extends into the cylinder interior but does not cross the cylinder.